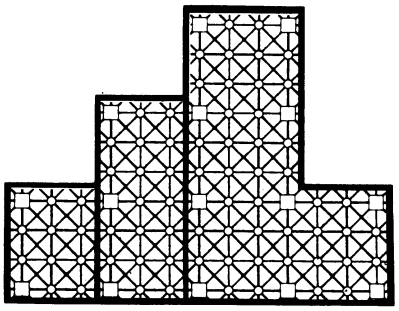
Systolic architectures, which permit multiple computations for each memory access, can speed execution of compute-bound problems without increasing I/O requirements.



Why Systolic Architectures?

H. T. Kung
Carnegie-Mellon University

High-performance, special-purpose computer systems are typically used to meet specific application requirements or to off-load computations that are especially taxing to general-purpose computers. As hardware cost and size continue to drop and processing requirements become well-understood in areas such as signal and image processing, more special-purpose systems are being constructed. However, since most of these systems are built on an ad hoc basis for specific tasks, methodological work in this area is rare. Because the knowledge gained from individual experiences is neither accumulated nor properly organized, the same errors are repeated. I/O and computation imbalance is a notable example—often, the fact that I/O interfaces cannot keep up with device speed is discovered only after constructing a high-speed, special-purpose device.

We intend to help correct this ad hoc approach by providing a general guideline—specifically, the concept of systolic architecture, a general methodology for mapping high-level computations into hardware structures. In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart. The system works like an automobile assembly line where different people work on the same car at different times and many cars are assembled simultaneously. An assembly line is always linear, however, and systolic systems are sometimes two-dimensional. They can be rectangular, triangular, or hexagonal to make use of higher degrees of parallelism. Moreover, to implement a variety of computations, data flow in a systolic system may be at multiple speeds in multiple directions—both inputs and (partial) results flow, whereas only results flow in classical pipelined systems. Generally speaking, a systolic system is easy to implement because of its regularity and easy to reconfigure (to meet various outside constraints) because of its modularity.

The systolic architectural concept was developed at Carnegie-Mellon University,[1-7] and versions of systolic processors are being designed and built by several industrial and governmental organizations.[8-10] This article reviews the basic principle of systolic architectures and explains why they should result in cost-effective, high-performance special-purpose systems for a wide range of problems.

Key architectural issues in designing special-purpose systems

Roughly, the cycle for developing a special-purpose system can be divided into three phases—task definition, design, and implementation. During task definition, some system performance bottleneck is identified, and a decision on whether or not to resolve it with special-purpose hardware is made. The evaluation required for task definition is most fundamental, but since it is often application-dependent, we will concentrate only on architectural issues related to the design phase and will assume routine implementation.

Simple and regular design. Cost-effectiveness has always been a chief concern in designing special-purpose systems; their cost must be low enough to justify their limited applicability. Costs can be classified as nonrecurring (design) and recurring (parts) costs. Part costs are dropping rapidly due to advances in integrated-circuit technology, but this advantage applies equally to both special-purpose and general-purpose systems. Furthermore, since special-purpose systems are seldom produced in large quantities, part costs are less important than design costs. Hence, the design cost of a special-purpose system must be relatively small for it to be more attractive than a general-purpose approach.

Fortunately, special-purpose design costs can be reduced by the use of appropriate architectures. If a structure can truly be decomposed into a few types of simple substructures or building blocks, which are used repetitively with simple interfaces, great savings can be achieved. This is especially true for VLSI designs where a single chip comprises hundreds of thousands of components. To cope with that complexity, simple and regular designs, similar

0018-9162/82/0100-0037$00.75

to some of the techniques used in constructing large software systems, are essential.[11] In addition, special-purpose systems based on simple, regular designs are likely to be modular and therefore adjustable to various performance goals—that is, system cost can be made proportional to the performance required. This suggests that meeting the architectural challenge for simple, regular designs yields cost-effective special-purpose systems.

Concurrency and communication. There are essentially two ways to build a fast computer system. One is to use fast components, and the other is to use concurrency. The last decade has seen an order of magnitude decrease in the cost and size of computer components but only an incremental increase in component speed.[12] With current technology, tens of thousands of gates can be put in a single chip, but no gate is much faster than its TTL counterpart of 10 years ago. Since the technological trend clearly indicates a diminishing growth rate for component speed, any major improvement in computation speed must come from the concurrent use of many processing elements. The degree of concurrency in a special-purpose system is largely determined by the underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. When a large number of processing elements work simultaneously, coordination and communication become significant—especially with VLSI technology where routing costs dominate the power, time, and area required to implement a computation.[13] The issue here is to design algorithms that support high degrees of concurrency, and in the meantime to employ only simple, regular communication and control to enable efficient implementation.

Balancing computation with I/O. Since a special-purpose system typically receives data and outputs results through an attached host, I/O considerations influence overall performance. (The host in this context can mean a computer, a memory, a real-time device, etc. In practice, the special-purpose system may actually input from one "physical" host and output to another.) The ultimate performance goal of a special-purpose system is—and should be no more than—a computation rate that balances the available I/O bandwidth with the host. Since an accurate a priori estimate of available I/O bandwidth in a complex system is usually impossible, the design of a special-purpose system should be modular so that its structure can be easily adjusted to match a variety of I/O bandwidths.

Suppose that the I/O bandwidth between the host and a special-purpose system is 10 million bytes per second, a rather high bandwidth for present technology. Assuming that at least two bytes are read from or written to the host for each operation, the maximum rate will be only 5 million operations per second, no matter how fast the special-purpose system can operate (see Figure 1). Orders of magnitude improvements on this throughput are possible only if multiple computations are performed per I/O access. However, the repetitive use of a data item requires it to be stored inside the system for a sufficient length of time. Thus, the I/O problem is related not only to the available I/O bandwidth, but also to the available memory internal to the system. The question then is how to arrange a computation together with an appropriate memory structure so that computation time is balanced with I/O time.

The I/O problem becomes especially severe when a large computation is performed on a small special-purpose system. In this case, the computation must be decomposed. Executing subcomputations one at a time may require a substantial amount of I/O to store or retrieve intermediate results. Consider, for example, performing the n-point fast Fourier transform using an S-point device when n is large and S is small. Figure 2 depicts the n-point FFT computation and a decomposition scheme for $n = 16$ and $S = 4$. Note that each subcomputation block is sufficiently small so that it can be handled by the 4-point device. During execution, results of a block must be temporarily sent to the host and later retrieved to be combined with results of other blocks as they become available. With the decomposition scheme shown in Figure 2b, the total number of I/O operations is $O(n \log n/\log S)$. In fact, it has been shown that, to perform the n-point FFT with a device of $O(S)$ memory, at least this many I/O operations are needed for any decomposition scheme.[14] Thus, for the n-point FFT problem, an S-point device cannot achieve more than an $O(\log S)$ speed-up ratio over the conventional $O(n \log n)$ software implementation time, and since it is a consequence of the I/O consideration, this upper bound holds independently of device speed. Similar upper bounds have been established for speed-up ratios achievable by devices for other computations such as sorting and matrix multiplication.[14,15] Knowing the I/O-imposed performance limit helps prevent overkill in the design of a special-purpose device.

In practice, problems are typically "larger" than special-purpose devices. Therefore, questions such as how a computation can be decomposed to minimize I/O, how the I/O requirement is related to the size of a special-purpose system and its memory, and how the I/O bandwidth limits the speed-up ratio achievable by a special-purpose system present another set of challenges to the system architect.

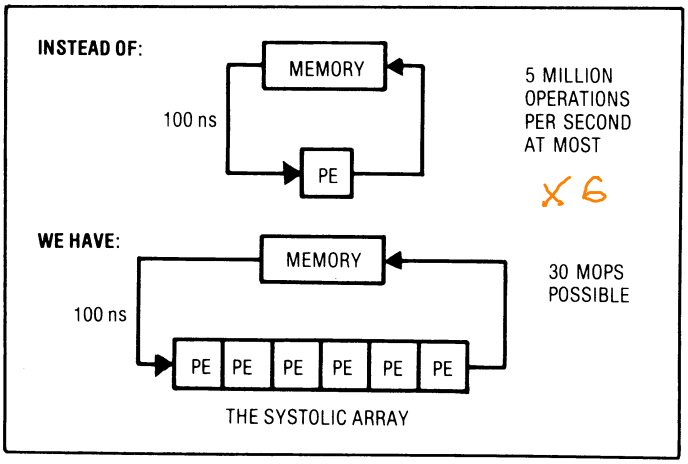


Figure 1. Basic principle of a systolic system.

Systolic architectures: the basic principle

As a solution to the above challenges, we introduce systolic architectures, an architectural concept originally proposed for VLSI implementation of some matrix operations.[5] Examples of systolic architectures follow in the next section, which contains a walk-through of a family of designs for the convolution computation.

A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Because simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the "boundary cells." For example, in a systolic array, only those cells on the array boundaries may be I/O ports for the system.

Computational tasks can be conceptually classified into two families—compute-bound computations and I/O-bound computations. In a computation, if the total number of operations is larger than the total number of input and output elements, then the computation is compute-bound, otherwise it is I/O-bound. For example, the ordinary matrix-matrix multiplication algorithm represents a compute-bound task, since every entry in a matrix is multiplied by all entries in some row or column of the other matrix. Adding two matrices, on the other hand, is I/O-bound, since the total number of adds is not larger than the total number of entries in the two matrices. It should be clear that any attempt to speed up an I/O-bound computation must rely on an increase in memory bandwidth. Memory bandwidth can be increased by the use of either fast components (which could be expensive) or interleaved memories (which could create complicated memory management problems). Speeding up a compute-bound computation, however, may often be accomplished in a relatively simple and inexpensive manner, that is, by the systolic approach.

The basic principle of a systolic architecture, a systolic array in particular, is illustrated in Figure 1. By replacing a single processing element with an array of PEs, or cells in the terminology of this article, a higher computation throughput can be achieved without increasing memory bandwidth. The function of the memory in the diagram is analogous to that of the heart; it "pulses" data (instead of blood) through the array of cells. The crux of this approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell it passes while being "pumped" from cell to cell along the array. This is possible for a wide class of compute-bound computations where multiple operations are performed on each data item in a repetitive manner.

Being able to use each input data item a number of times (and thus achieving high computation throughput with only modest memory bandwidth) is just one of the many advantages of the systolic approach. Other advantages, such as modular expansibility, simple and regular data and control flows, use of simple and uniform cells, elimination of global broadcasting, and fan-in and (possibly) fast response time, will be illustrated in various systolic designs in the next section.

A family of systolic designs for the convolution computation

To provide concrete examples of various systolic structures, this section presents a family of systolic designs for the convolution problem, which is defined as follows:

Given the sequence of weights $\{w_1, w_2, \ldots, w_k\}$ and the input sequence $\{x_1, x_2, \ldots, x_n\}$,

compute the result sequence $\{y_1, y_2, \ldots, y_{n+1-k}\}$ defined by

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}$$

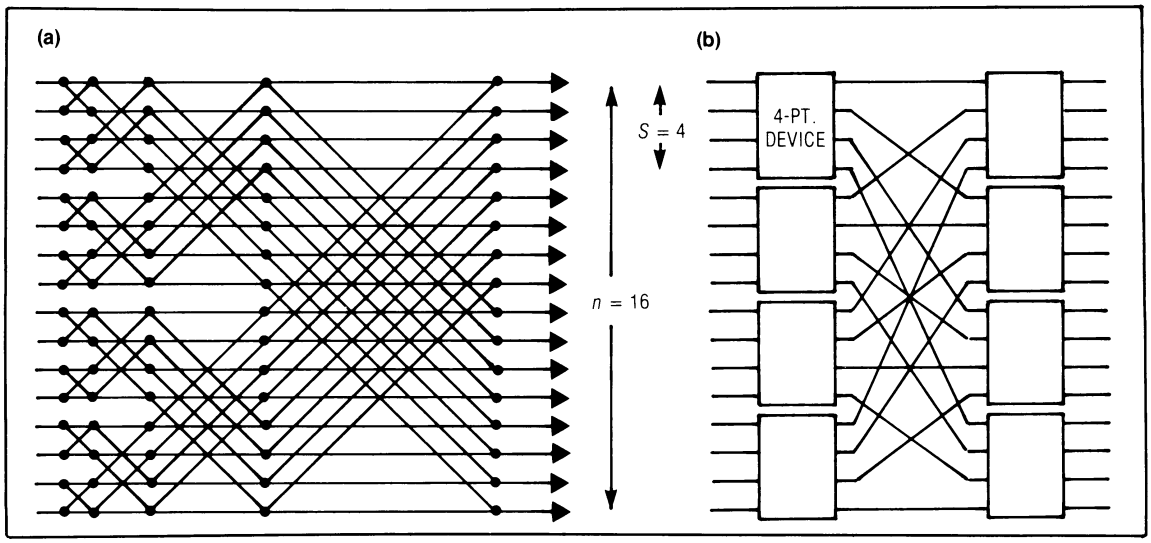


Figure 2. (a) 16-point fast-Fourier-transform graph; (b) decomposing the FFT computation with $n = 16$ and $S = 4$.

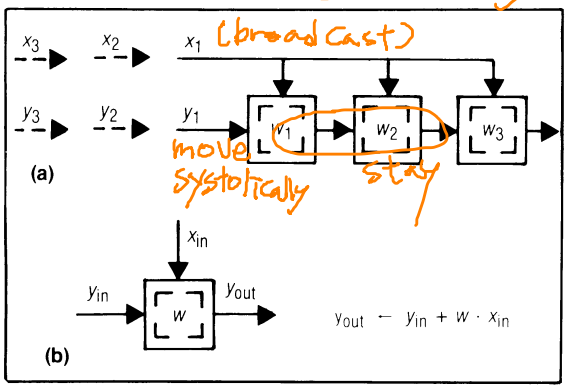


Figure 3. Design B1: systolic convolution array (a) and cell (b) where x_i's are broadcast, w_i's stay, and y_i's move systolically.

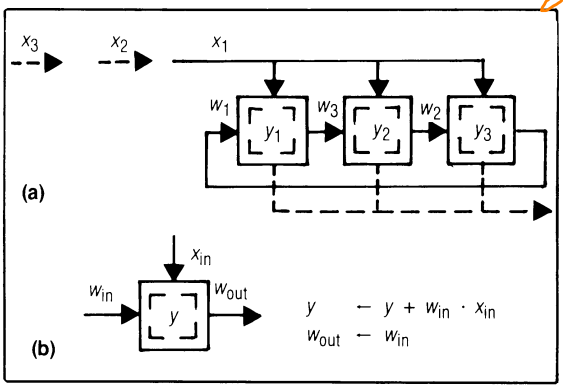


Figure 4. Design B2: systolic convolution array (a) and cell (b) where x_i's are broadcast, y_i's stay, and w_i's move systolically.

We consider the convolution problem because it is a simple problem with a variety of enlightening systolic solutions, because it is an important problem in its own right, and more importantly, because it is representative of a wide class of computations suited to systolic designs. The convolution problem can be viewed as a problem of combining two data streams, w_i's amd x_i's, in a certain manner (for example, as in the above equation) to form a resultant data stream of y_i's. This type of computation is common to a number of computation routines, such as filtering, pattern matching, correlation, interpolation, polynomial evaluation (including discrete Fourier transforms), and polynomial multiplication and division. For example, if multiplication and addition are interpreted as comparison and boolean AND, respectively, then the convolution problem becomes the pattern matching problem.[1] Architectural concepts for the convolution problem can thus be applied to these other problems as well.

The convolution problem is compute-bound, since each input x_i is to be multiplied by each of the k weights. If the x_i is input separately from memory for each multiplication, then when k is large, memory bandwidth becomes a bottleneck, precluding a high-performance solution. As indicated earlier, a systolic architecture resolves this I/O bottleneck by making multiple use of each x_i fetched from the memory. Based on this principle, several systolic designs for solving the convolution problem are described below. For simplicity, all illustrations assume that $k = 3$.

(Semi-) systolic convolution arrays with global data communication. If an x_i, once brought out from the memory, is broadcast to a number of cells, then the same x_i can be used by all the cells. This broadcasting technique is probably one of the most obvious ways to make multiple use of each input element. The opposite of broadcasting is fan-in, through which data items from a number of cells can be collected. The fan-in technique can also be used in a straightforward manner to resolve the I/O bottleneck problem. In the following, we describe systolic designs that utilize broadcasting and fan-in.

Design B1—broadcast inputs, move results, weights stay. The systolic array and its cell definition are depicted in Figure 3. Weights are preloaded to the cells, one at each cell, and stay at the cells throughout the computation. Partial results y_i move systolically from cell to cell in the left-to-right direction, that is, each of them moves over the cell to its right during each cycle. At the beginning of a cycle, one x_i is broadcast to all the cells and one y_i, initialized as zero, enters the left-most cell. During cycle one, $w_1 x_1$ is accumulated to y_1 at the left-most cell, and during cycle two, $w_1 x_2$ and $w_2 x_2$ are accumulated to y_2 and y_1 at the left-most and middle cells, respectively. Starting from cycle three, the final (and correct) values of $y_1, y_2, \ldots$ are output from the right-most cell at the rate of one y_i per cycle. The basic principle of this design was previously proposed for circuits to implement a pattern matching processor[16] and for circuits to implement polynomial multiplication.[17-20]

Design B2—broadcast inputs, move weights, results stay. In design B2 (see Figure 4), each y_i stays at a cell to accumulate its terms, allowing efficient use of available multiplier-accumulator hardware. (Indeed, this design is described in an application booklet for the TRW multiplier-accumulator chips.[21] The weights circulate around the array of cells, and the first weight w_1 is associated with a tag bit that signals the accumulator to output and resets its contents.* In design B1 (Figure 3), the systolic path for moving y_i's may be considerably wider than that for moving w_i's in design B2 because for numerical accuracy y_i's typically carry more bits than w_i's. The use of multiplier-accumulators in design B2 may also help increase precision of the results, since extra bits can be kept in these accumulators with modest cost. Design B1, however, does have the advantage of not requiring a separate bus (or other global network), denoted by a dashed line in Figure 4, for collecting outputs from individual cells.

Design F—fan-in results, move inputs, weights stay. If we consider the vector of weights $(w_k, w_{k-1}, \ldots, w_1)$ as being fixed in space and input vector $(x_n, x_{n-1}, \ldots, x_1)$ as sliding over the weights in the left-to-right direction, then the convolution problem is one that computes the inner product of the weight vector and the section of input vector it overlaps. This view suggests the systolic array

*To avoid complicated pictures, control structures such as the use of tag bits to gate outputs from cells are omitted from the diagrams of this article.

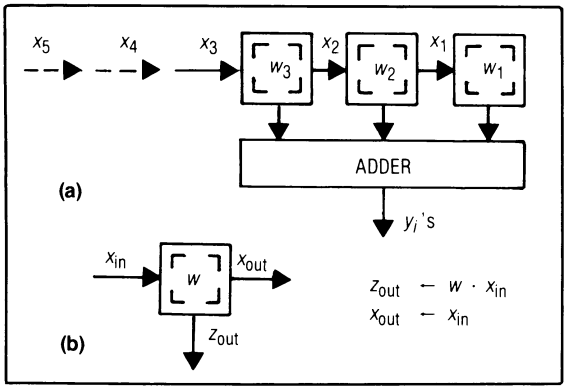


Figure 5. Design F: systolic convolution array (a) and cell (b) where w_i's stay, x_i's move systolically, and y_i's are formed through the fan-in of results from all the cells.

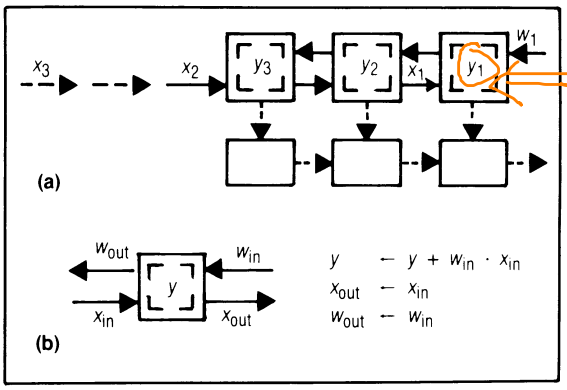


Figure 6. Design R1: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and y_i's move in opposite directions systolically.

shown in Figure 5. Weights are preloaded to the cells and stay there throughout the computation. During a cycle, all x_i's move one cell to the right, multiplications are performed at all cells simultaneously, and their results are fanned-in and summed using an adder to form a new y_i. When the number of cells, k, is large, the adder can be implemented as a pipelined adder tree to avoid large delays in each cycle. Designs of this type using unbounded fan-in have been known for quite a long time, for example, in the context of signal processing[33] and in the context of pattern matching.[43]

(Pure-) systolic convolution arrays without global data communication. Although global broadcasting or fan-in solves the I/O bottleneck problem, implementing it in a modular, expandable way presents another problem. Providing (or collecting) a data item to (or from) all the cells of a systolic array, during each cycle, requires the use of a bus or some sort of tree-like network. As the number of cells increases, wires become long for either a bus or tree structure; expanding these non-local communication paths to meet the increasing load is difficult without slowing down the system clock. This engineering difficulty of extending global networks is significant at chip, board, and higher levels of a computer system. Fortunately, as will be demonstrated below, systolic convolution arrays without global data communication do exist. Potentially, these arrays can be extended to include an arbitrarily large number of cells without encountering engineering difficulties (the problem of synchronizing a large systolic array is discussed later).

Design R1—results stay, inputs and weights move in opposite directions. In design R1 (see Figure 6) each partial result y_i stays at a cell to accumulate its terms. The x_i's and w_i's move systolically in opposite directions such that when an x meets a w at a cell, they are multiplied and the resulting product is accumulated to the y staying at that cell. To ensure that each x_i is able to meet every w_i, consecutive x_i's on the x data stream are separated by two cycle times and so are the w_i's on the w data stream.

Like design B2, design R1 can make efficient use of available multiplier-accumulator hardware; it can also use a tag bit associated with the first weight, w_1, to trigger the output and reset the accumulator contents of a cell.

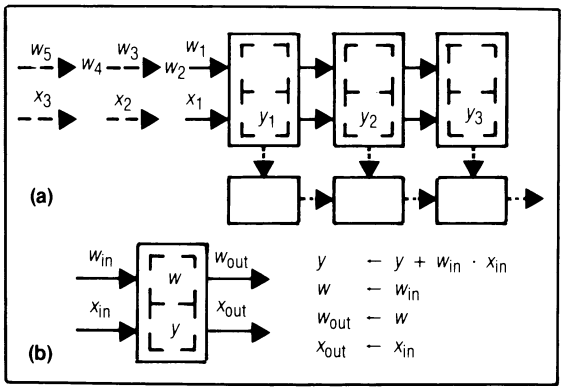


Figure 7. Design R2: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and w_i's both move in the same direction but at different speeds.

Design R1 has the advantage that it does not require a bus, or any other global network, for collecting output from cells; a systolic output path (indicated by broken arrows in Figure 6) is sufficient. Because consecutive w_i's are well separated by two cycle times, a potential conflict—that more than one y_i may reach a single latch on the systolic output path simultaneously—cannot occur. It can also be easily checked that the y_i's will output from the systolic output path in the natural ordering $y_1, y_2, \ldots$. The basic idea of this design, including that of the systolic output path, has been used to implement a pattern matching chip.[1]

Notice that in Figure 6 only about one-half the cells are doing useful work at any time. To fully utilize the potential throughput, two independent convolution computations can be interleaved in the same systolic array, but cells in the array would have to be modified slightly to support the interleaved computation. For example, an additional accumulator would be required at each cell to hold a temporary result for the other convolution computation.

Design R2—results stay, inputs and weights move in the same direction but at different speeds. One version of design R2 is illustrated in Figure 7. In this case both the x and w data streams move from left to right systolically, but the x_i's move twice as fast as the w_i's. More precisely,

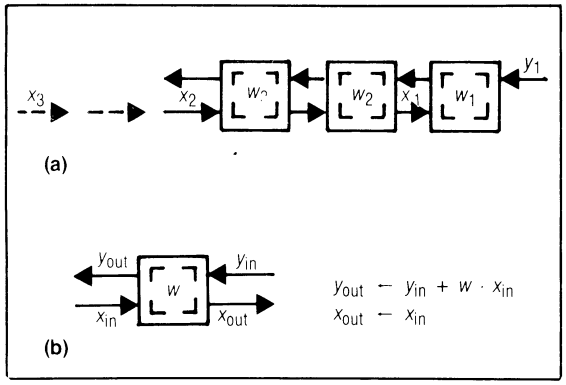


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's move systolically in opposite directions.

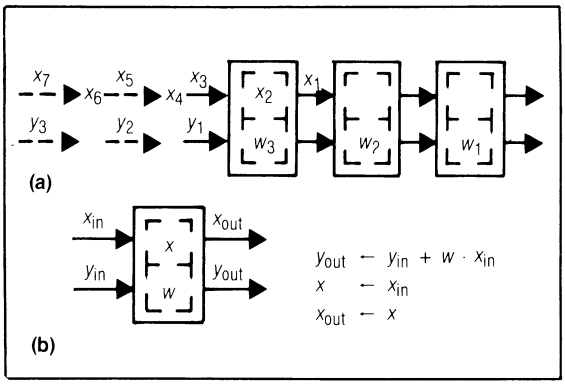


Figure 9. Design W2: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's both move systolically in the same direction but at different speeds.

each w_i stays inside every cell it passes for one extra cycle, thus taking twice as long to move through the array as any x_i. In this design, multiplier-accumulator hardware can be used effectively and so can the tag bit method to signal the output of the accumulator contents at each cell. Compared to design R1, this design has the advantage that all cells work all the time when performing a single convolution, but it requires an additional register in each cell to hold a w value. This algorithm has been used for implementing a pipeline multiplier.[22]

There is a dual version of design R2; we can have the w_i's move twice as fast as the x_i's. To create delays for the x data stream, this dual design requires a register in each cell for storing an x rather than a w value. For circumstances where the w_i's carry more bits than the x_i's, the dual design becomes attractive.

Design W1—weights stay, inputs and results move in opposite directions. In design W1 (and design W2, below), weights stay, one at each cell, but results and inputs move systolically. These designs are not geared to the most effective use of available multiplier-accumulator hardware, but for some other circumstances they are potentially more efficient than the other designs. Because the same set of weights is used for computing all the y_i's and different sets of the x_i's are used for computing different y_i's, it is natural to have the w_i's preloaded to the cells and stay there, and let the x_i's and the y_i's move along the array. We will see some advantages of this arrangement in the systolic array depicted in Figure 8, which is a special case of a proposed systolic filtering array.[3] This design is fundamental in the sense that it can be naturally extended to perform recursive filtering[2,3] and polynomial division.[23]

In design W1, the w_i's stay and the x_i's and y_i's move systolically in opposite directions. Similar to design R1, consecutive x_i's and y_i's are separated by two cycle times. Note that because the systolic path for moving the y_i's already exists, there is no need for another systolic output path as in designs R1 and R2. Furthermore, for each i, y_i outputs from the left-most cell during the same cycle as its last input, x_{i+k-1} (or x_{i+2} for $k = 3$), enters that cell. Thus, this systolic array is capable of outputting a y_i every two cycle times with constant response time. Design W1, however, suffers from the same drawback as design R1, namely, only approximately one-half the cells work at any given time unless two independent convolution computations are interleaved in the same array. The next design, like design R2, overcomes this shortcoming by having both the x_i's and y_i's move in the same direction but at different speeds.

Design W2—weights stay, inputs and results move in the same direction but at different speeds. With design W2 (Figure 9) all the cells work all the time, but it loses one advantage of design W1, the constant response time. The output of y_i now takes place k cycles after the last of its inputs starts entering the left-most cell of the systolic array. This design has been extended to implement 2-D convolutions,[6,24] where high throughputs rather than fast responses are of concern. Similar to design R1, design W2 has a dual version for which the x_i's move twice as fast as the y_i's.

Remarks. The designs presented above by no means exhaust all the possible systolic designs for the convolution problem. For example, it is possible to have systolic designs where results, weights, and inputs all move during each cycle. It could also be advantageous to include inside each cell a "cell memory" capable of storing a set of weights. With this feature, using a systolic control (or address) path, weights can be selected on-the-fly to implement interpolation or adaptive filtering.[24] Moreover, the flexibility introduced by the cell memories and systolic control can make the same systolic array implement different functions. Indeed, the ESL systolic processor[8,10] utilizes cell memories to implement multiple functions including convolution and matrix multiplication.

Once one systolic design is obtained for a problem, it is likely that a set of other systolic designs can be derived similarly. The challenge is to understand precisely the strengths and drawbacks of each design so that an appropriate design can be selected for a given environment. For example, if there are more weights than cells, it's useful to know that a scheme where partial results stay generally requires less I/O than one where partial results move, since the latter scheme requires partial results to be input and output many times. A single multiplier-accumulator hardware component often represents a cost-effective implementation of the multiplier and adder

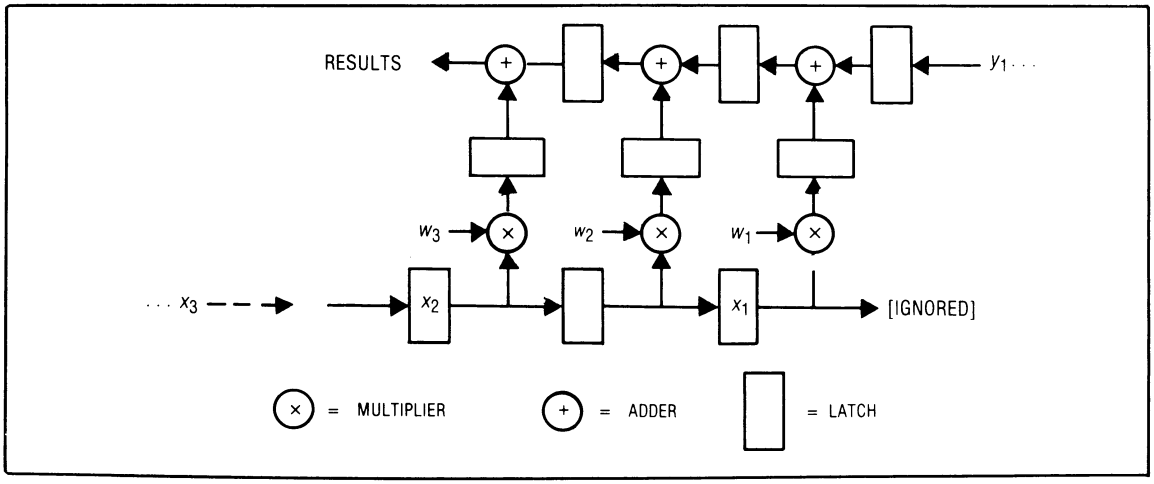


Figure 10. Overlapping the executions of multiply and add in design W1.

needed by each cell of a systolic convolution array. However, for improving throughput, sometimes it may be worthwhile to implement multiplier and adder separately to allow overlapping of their executions. Figure 10 depicts such a modification to design W1. Similar modifications can be made to other systolic convolution arrays. Another interesting scenario is the following one. Suppose that one or several cells are to be implemented directly with a single chip and the chip pin bandwidth is the implementation bottleneck. Then, since the basic cell of some semi-systolic convolution arrays such as designs B1 and F require only three I/O ports, while that of a pure-systolic convolution array always requires four, a semi-systolic array may be preferable for saving pins, despite the fact that it requires global communication.

Criteria and advantages

Having described a family of systolic convolution arrays, we can now be more precise in suggesting and evaluating criteria for the design of systolic structures.

(1) *The design makes multiple use of each input data item.* Because of this property, systolic systems can achieve high throughputs with modest I/O bandwidths for outside communication. To meet this criterion, one can either use global data communications, such as broadcast and unbounded fan-in, or have each input travel through an array of cells so that it is used at each cell. For modular expansibility of the resulting system, the second approach is preferable.

(2) *The design uses extensive concurrency.* The processing power of a systolic architecture comes from concurrent use of many simple cells rather than sequential use of a few powerful processors as in many conventional architectures. Concurrency can be obtained by pipelining the stages involved in the computation of each single result (for example, design B1), by multiprocessing many results in parallel (designs R1 and R2), or by both. For some designs, such as W1, it is possible to completely overlap I/O and computation times to further increase concurrency and provide constant-time responses.

To a given problem there could be both one- and two-dimensional systolic array solutions. For example, two-dimensional convolution can be performed by a one-dimensional systolic array[24,25] or a two-dimensional systolic array.[6] When the memory speed is more than cell speed, two-dimensional systolic arrays such as those depicted in Figure 11 should be used. At each cell cycle, all the I/O ports on the array boundaries can input or output data items to or from the memory; as a result, the available memory bandwidth can be fully utilized. Thus, the choice of a one- or two-dimensional scheme is very dependent on how cells and memories will be implemented.

As in one-dimensional systolic arrays, data in two-dimensional arrays may flow in multiple directions and at multiple speeds. For examples of two-dimensional systolic arrays, see Guibas et al.[26] and Kung and Lehman[4] (type R), Kung and Leiserson[5] and Weiser and Davis[27] (type H), and Bojanczyk et al.[28] and Gentleman and Kung[29] (type T). In practice, systolic arrays can be chained together to form powerful systems such as the one depicted in Figure 12, which is capable of producing on-the-fly the least-squares fit to all the data that have arrived up to any given moment.[29]

For the systolic structures discussed in the preceding section, computations are pipelined over an array of cells. To permit even higher concurrency, it is sometimes possible to introduce another level of pipelining by allowing the operations inside the cells themselves to be pipelined. (Note that pipelined arithmetic units will become increasingly common as VLSI makes the extra circuits needed for

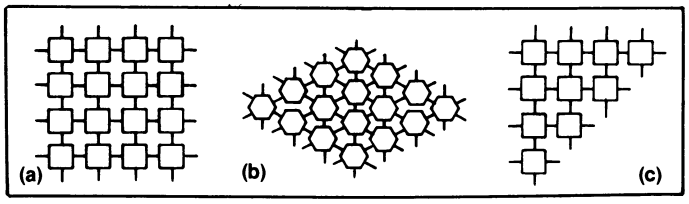


Figure 11. Two-dimensional systolic arrays: (a) type R, (b) type H, and (c) type T.

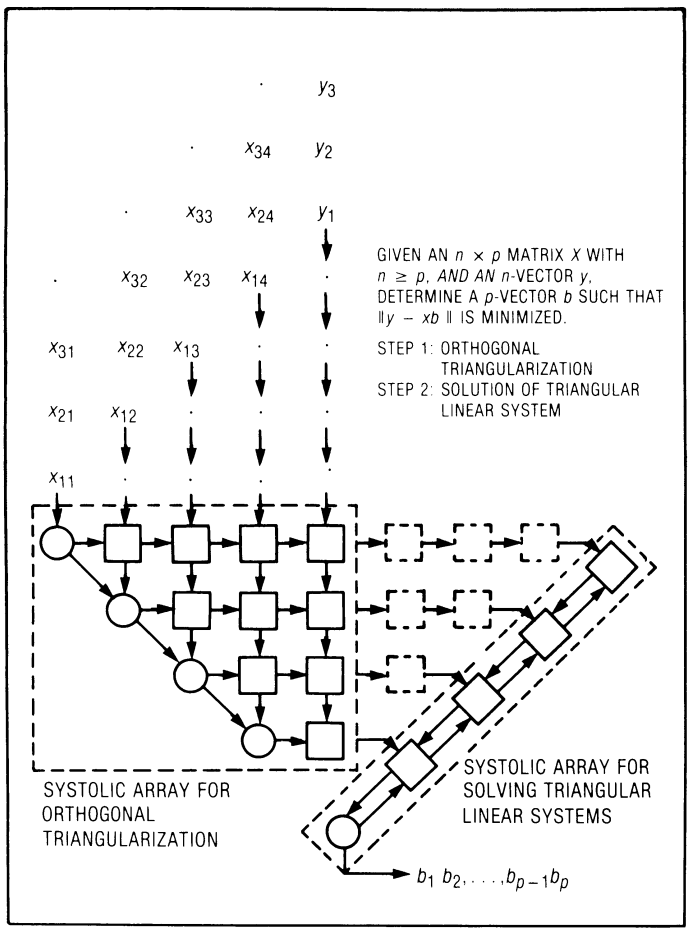


Figure 12. On-the-fly least-squares solutions using one- and two-dimensional systolic arrays, with $p = 4$.

staging affordable.) Both designs W1 and W2 support two-level pipelining.[25] Since system cycle time is the time of a stage of a cell, rather than the whole cell cycle time, two-level pipelined systolic systems significantly improve throughput.

(3) *There are only a few types of simple cells.* To achieve performance goals, a systolic system is likely to use a large number of cells. The cells must be simple and of only a few types to curtail design and implementation costs, but exactly how simple is a question that can only be answered on a case by case basis. For example, if a systolic system consisting of many cells is to be implemented on a single chip, each cell should probably contain only simple logic circuits plus a few words of memory On the other hand, for board implementations each cell could reasonably contain a high-performance arithmetic unit plus a few thousand words of memory. There is, of course, always a trade-off between cell simplicity and flexibility.

(4) *Data and control flows are simple and regular.* Pure systolic systems totally avoid long-distance or irregular wires for data communication. The only global communication (besides power and ground) is the system clock. Of course, self-timed schemes can be used instead for synchronizing neighboring cells, but efficient implementations of self-timed protocols may be difficult. Fortunately, for any one-dimensional systolic array, a global clock parallel to the array presents no problems, even if the array is arbitrarily long. The systolic array (with data flowing in either one or opposite directions) will operate correctly despite the possibility of a large clock skew between its two ends.[30] However, large two-dimensional arrays may require slowdown of the global clock to compensate for clock skews. Except for this possible problem in the two-dimensional case, systolic designs are completely modular and expandable; they present no difficult synchronization or resource conflict problems. Software overhead associated with operations such as address indexing are totally eliminated in systolic systems. This advantage alone can mean a substantial performance improvement over conventional general-purpose computers. Simple, regular control and communication also imply simple, area-efficient layout or wiring—an important advantage in VLSI implementation.

In summary, systolic designs based on these criteria are *simple* (a consequence of properties 3 and 4), *modular* and *expandable* (property 4), and yield *high performance* (properties 1, 2, and 4). They therefore meet the architectural challenges for special-purpose systems. A unique characteristic of the systolic approach is that as the number of cells expands the system cost and performance increase proportionally, provided that the size of the underlying problem is sufficiently large. For example, a systolic convolution array can use an arbitrarily large number of cells cost-effectively, if the kernel size (that is, the number of weights) is large. This is in contrast to other parallel architectures which are seldom cost-effective for more than a small number of processors. From a user's point of view, a systolic system is easy to use—he simply pumps in the input data and then receives the results either on-the-fly or at the end of the computation.

Summary and concluding remarks

Bottlenecks to speeding up a computation are often due to limited system memory bandwidths, so called *von Neumann bottlenecks,* rather than limited processing capabilities per se. This problem can certainly be expected for I/O-bound computations, but with a conventional architectural approach, it may be present even for compute-bound computations. For every operation, at least one or two operands have to be fetched (or stored) from (or to) memory, so the total amount of I/O is proportional to the number of operations rather than the number of inputs and outputs. Thus, a problem that was originally compute-bound can become I/O-bound during its execution. This unfortunate situation is the result of a mismatch between the computation and the architecture. Systolic architectures, which ensure multiple computations per memory access, can speed up compute-bound computations without increasing I/O requirements.

The convolution problem is just one of many compute-bound computations that can benefit from the systolic approach. Systolic designs using (one- or two-dimensional) array or tree structures are available for the following regular, compute-bound computations.

Signal and image processing:

- FIR, IIR filtering, and 1-D convolution[2,3,31];

- 2-D convolution and correlation[6,8,10,24,25];
- discrete Fourier transform[2,3];
- interpolation[24];
- 1-D and 2-D median filtering[32]; and
- geometric warping.[24]

Matrix arithmetic:
- matrix-vector multiplication[5];
- matrix-matrix multiplication[5,27];
- matrix triangularization (solution of linear systems, matrix inversion)[5,29];
- QR decomposition (eigenvalue, least-square computations)[28,29]; and
- solution of triangular linear systems.[5]

Non-numeric applications:
- data structures—stack and queue,[34] searching,[15,35,36] priority queue,[7] and sorting[7,15];
- graph algorithms—transitive closure,[26] minimum spanning trees,[37] and connected components[38];
- language recognition—string matching[1] and regular expression[39];
- dynamic programming[26];
- encoders (polynomial division)[23]; and
- relational data-base operations.[4,40]

In general, systolic designs apply to any compute-bound problem that is regular—that is, one where repetitive computations are performed on a large set of data. Thus, the above list is certainly not complete (and was not intended to be so). Its purpose is to provide a range of typical examples and possible applications. After studying several of these examples, one should be able to start designing systolic systems for one's own tasks. Some systolic solutions can usually be found without too much difficulty. (I know of only one compute-bound problem that arises naturally in practice for which no systolic solution is known, and I cannot prove that a systolic solution is impossible.) This is probably due to the fact that most compute-bound problems are inherently regular in the sense that they are definable in terms of simple recurrences. Indeed, the notion of systolicity is implicit in quite a few previously known special-purpose designs, such as the sorting[41] and multiply designs.[22] This should not come as a surprise; as we have been arguing systolic structures are essential for obtaining any cost-effective, high-performance solution to compute-bound problems. It is useful, however, to make the systolic concept explicit so that designers will be conscious of this important design criterion.

While numerous systolic designs are known today, the question of their automatic design is still open. But recent efforts show significant progress.[27,42] Leiserson and Saxe, for instance, can convert some semi-systolic systems involving broadcasting or unbounded fan-in into pure-systolic systems without global data communication.[42] A related open problem concerns the specification and verification of systolic structures. For implementation and proof purposes, rigorous notation other than informal pictures (as used in this article) for specifying systolic designs is desirable.

With the development of systolic architectures, more and more special-purpose systems will become feasible—especially systems that implement fixed, well-understood computation routines. But the ultimate goal is effective use of systolic processors in general computing environments to off-load regular, compute-bound computations. To achieve this goal further research is needed in two areas. The first concerns the system integration: we must provide a convenient means for incorporating high-performance systolic processors into a complete system and for understanding their effective utilization from a system point of view. The second research area is to specify building-blocks for a variety of systolic processors so that, once built, these building blocks can be programmed to form basic cells for a number of systolic systems. The building-block approach seems inherently suitable to systolic architectures since they tend to use only a few types of simple cells. By combining these building-blocks regularly, systolic systems geared to different applications can be obtained with little effort. ■

Acknowledgment

Parts of this article were written while the author was on leave from Carnegie-Mellon University with ESL's Advanced Processor Technology Laboratory in San Jose, California, January-August 1981. Most of the research reported here was carried out at CMU and was supported in part by the Office of Naval Research under Contracts N00014-76-C-0370, NR 044-422 and N00014-80-C-0236, NR 048-659, in part by the National Science Foundation under Grant MCS 78-236-76, and in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

References

1. M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *Computer,* Vol. 13, No. 1, Jan. 1980, pp. 26-40.
2. H. T. Kung, "Let's Design Algorithms for VLSI Systems," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, Jan. 1979, pp. 65-90.
3. H. T. Kung, "Special-Purpose Devices for Signal and Image Processing: An Opportunity in VLSI," *Proc. SPIE, Vol. 241, Real-Time Signal Processing III,* Society of Photo-Optical Instrumentation Engineers, July 1980, pp. 76-84.
4. H. T. Kung and P. L. Lehman, "Systolic (VLSI) Arrays for Relational Database Operations," *Proc. ACM-Sigmod 1980 Int'l Conf. Management of Data,* May 1980, pp. 105-116.
5. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc. 1978,* Society for Industrial and Applied Mathematics, 1979, pp. 256-282.
6. H. T. Kung and S. W. Song, *A Systolic 2-D Convolution Chip,* Technical Report CMU-CS-81-110, Carnegie-Mellon University Computer Science Dept., Mar. 1981.
7. C. E. Leiserson, "Systolic Priority Queues," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, Jan. 1979, pp. 199-214.
8. J. Blackmer, P. Kuekes, and G. Frank, "A 200 MOPS systolic processor," *Proc. SPIE, Vol. 298, Real-Time Signal Processing IV,* Society of Photo-Optical Instrumentation Engineers, 1981.

9. K. Bromley, J. J. Symanski, J. M. Speiser, and H. J. Whitehouse, "Systolic Array Processor Developments," in *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, and G. L. Steele, Jr., (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 273-284.
10. D. W. L. Yen and A. V. Kulkarni, "The ESL Systolic Processor for Signal and Image Processing," *Proc. 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* Nov. 1981, pp. 265-272.
11. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.
12. R. N. Noyce, "Hardware Prospects and Limitations," in *The Computer Age: A Twenty-Year Review,* M. L. Dertouzos and J. Moses (eds.), IEEE Press, 1979, pp. 321-337.
13. I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," *Scientific American,* Vol. 237, No. 3, Sept. 1977, pp. 210-228.
14. J-W. Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. 13th Annual ACM Symp. Theory of Computing,* ACM Sigact, May 1981, pp. 326-333.
15. S. W. Song, *On a High-Performance VLSI Solution to Database Problems,* PhD dissertation, Carnegie-Mellon University, Computer Science Dept., July 1981.
16. A. Mukhopadhyay, "Hardware Algorithms for Nonnumeric Computation," *IEEE Trans. Computers,* Vol. C-28, No. 6, June 1979, pp. 384-394.
17. D. Cohen, *Mathematical Approach to Computational Networks,* Technical Report ISI/RR-78-73, University of Southern California, Information Sciences Institute, 1978.
18. D. A. Huffman, "The Synthesis of Linear Sequential Coding Networks," in *Information Theory,* C. Cherry (ed.), Academic Press, 1957, pp. 77-95.
19. K. Y. Liu, "Architecture for VLSI Design of Reed-Solomon Encoders," *Proc. Second Caltech VLSI Conf.* Jan. 1981.
20. W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes,* MIT Press, Cambridge, Mass., 1972.
21. L. Schirm IV, *Multiplier-Accumulator Application Notes,* TRW LSI Products, Jan. 1980.
22. R. F. Lyon, "Two's Complement Pipeline Multipliers," *IEEE Trans. Comm.,* Vol. COM-24, No. 4, Apr. 1976, pp. 418-425.
23. H. T. Kung, "Use of VLSI in Algebraic Computation: Some Suggestions," *Proc. 1981 ACM Symp. Symbolic and Algebraic Computation,* ACM Sigsam, Aug. 1981, pp. 218-222.
24. H. T. Kung and R.L. Picard, "Hardware Pipelines for Multi-Dimensional Convolution and Resampling," *Proc. 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* Nov. 1981, pp. 273-278.
25. H. T. Kung, L. M. Ruane, and D. W. L. Yen, "A Two-Level Pipelined Systolic Array for Convolutions," in *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, and G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 255-264.
26. L. J. Guibàs, H. T. Kung, and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, Jan. 1979, pp. 509-525.
27. U. Weiser and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," in *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, and G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 226-234.
28. A. Bojanczyk, R. P. Brent, and H. T. Kung, *Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh-Connected Processors,* Technical Report, Carnegie-Mellon University, Computer Science Dept. May 1981.
29. W. M. Gentleman and H. T. Kung, "Matrix Triangularization by Systolic Arrays," *Proc. SPIE, Vol. 298, Real-Time Signal Processing IV,* Society of Photo-optical Instrumentation Engineers, 1981.
30. A. Fisher and H. T. Kung, CMU Computer Science Dept. technical report, Jan. 1982.
31. P. R. Cappello and K. Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," in *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, and G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 245-254.
32. A. Fisher, "Systolic Algorithms for Running Order Statistics in Signal and Image Processing," in *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 265-272.
33. E. E. Swartzlander, Jr., and B. K. Gilbert, "Arithmetic for Ultra-High-Speed Tomography," *IEEE Trans. Computers,* Vol. C-29, No. 5, May, 1980, pp. 341-354.
34. L. J. Guibas and F. M. Liang, "Systolic Stacks, Queues, and Counters," *Proc. Conf. Advanced Research in VLSI,* MIT, Jan. 1982.
35. J. L. Bentley and H. T. Kung, "A Tree Machine for Searching Problems," *Proc. 1979 Int'l Conf. Parallel Processing,* Aug. 1979, pp. 257-266. Also available as a Carnegie-Mellon University Computer Science Dept. technical report, Aug. 1979.
36. T. Ottmann, A. L. Rosenberg, and L. J. Stockmeyer, "A Dictionary Machine (for VLSI)," Technical Report RC 9060 (#39615), IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1981.
37. J. L. Bentley, *A Parallel Algorithm for Constructing Minimum Spanning Trees," Journal of Algorithms,* Jan. 1980, pp. 51-59.
38. C. Savage, "A Systolic Data Structure Chip for Connectivity Problems," in *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, and G. L. Steele, Jr., (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 296-300.
39. M. J. Foster and H. T. Kung, "Recognize Regular Languages With Programmable Building-Blocks," in *VLSI 81,* Academic Press, Aug. 1981, pp. 75-84.
40. P. L. Lehman, "A Systolic (VLSI) Array for Processing Simple Relational Queries," in *VLSI Systems and Computations,* H. T Kung, R. F. Sproull, and G. I. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 285-295.
41. S. Todd, "Algorithm and Hardware for a Merge Sort Using Multiple Processors," *IBM J. Research and Development,* Vol. 22, No. 5, 1978, pp. 509-517.
42. C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *Proc. 22nd Annual Symp. Foundations of Computer Science,* IEEE Computer Society, Oct. 1981, pp. 23-36.
43. C. A. Mead et al., "128-Bit Multicomparator," *IEEE J. Solid-State Circuits,* Vol. SC-11, No. 5, Oct. 1976, pp. 692-695.

H. T. Kung is an associate professor of computer science at Carnegie-Mellon University, where he leads a research group in the design and implementation of high-performance VLSI systems. From January to September 1981 he was an architecture consultant to ESL, Inc., a subsidiary of TRW. His research interests are in paradigms of mapping algorithms and applications directly on chips and in theoretical foundations of VLSI computations. He serves on the editorial board of the *Journal of Digital Systems* and is the author of over 50 technical papers in computer science.

Kung graduated from National Tsing-Hua University, Taiwan, in 1968, and received his PhD from Carnegie-Mellon University in 1974.